Chapter 8

Decision Making

In This Chapter

- Comparing conditions with `if`
- Using comparison operators
- Adding `else` to the decision
- Creating an `if-else-if-else` structure
- Making logical decisions
- Working with a `switch-case` structure
- Appreciating the ternary operator

Decision making is the part of programming that makes you think a computer is smart. It's not, of course, but you can fool anyone by crafting your code to carry out directions based on certain conditions or comparisons. The process is really simple to understand, but deriving that understanding from the weirdo way it looks in a C program is why this chapter is necessary.

If What?

All human drama is based on disobedience. No matter what the rules, no matter how strict the guidelines, some joker breaks free and the rest is an interesting story. That adventure begins with the simple human concept of "what if." It's the same concept used for decision making in your programs, though in that instance only the word *if* is required.

Making a simple comparison

You make comparisons all the time. What will you wear in the morning? Should you avoid Bill's office because the receptionist says he's "testy" today? And how much longer will you put off going to the dentist? The computer is no different, albeit the comparisons it makes use values, not abstracts. (See Listing 8-1.)

Listing 8-1: A Simple Comparison

```
#include <stdio.h>

int main()
{
    int a,b;

    a = 6;
    b = a - 2;

    if( a > b)
    {
        printf("%d is greater than %d\n",a,b);
    }
    return(0);
}
```

Exercise 8-1: Create a new project using the source code shown in Listing 8-1. Build and run. Here's the output you should see:

```
6 is greater than 4
```

Fast and smart, that's what a computer is. Here's how the code works:

Line 5 declares two integer variables: `a` and `b`. The variables are assigned values in Lines 7 and 8, with the value of variable `b` being calculated to be 2 less than variable `a`.

Line 10 makes a comparison:

```
if( a > b)
```

Programmers read this line as, "If *a* is greater than *b*." Or when they're teaching the C language, they say, "If variable `a` is greater than variable `b`." And, no, they don't read the parentheses.

Lines 11 through 13 belong to the `if` statement. The meat in the sandwich is Line 12; the braces (curly brackets) don't play a decision-making role, other than hugging the statement at Line 12. If the comparison in Line 10 is true, the statement in Line 12 is executed. Otherwise, all statements between the braces are skipped.

Exercise 8-2: Edit the source code from Listing 8-1 so that addition instead of subtraction is performed in Line 8. Can you explain the program's output?

Introducing the if keyword

The `if` keyword is used to make decisions in your code based upon simple comparisons. Here's the basic format:

```
if(evaluation)
{
    statement;
}
```

The *`evaluation`* is a comparison, a mathematical operation, the result of a function or some other condition. If the condition is true, the *`statements`* (or *`statement`*) enclosed in braces are executed; otherwise, they're skipped.

- The `if` statement's evaluation need not be mathematical. It can simply be a function that returns a true or false value; for example:

  ```
  if(ready())
  ```

  This statement evaluates the return of the `ready()` function. If the function returns a true value, the statements belonging to `if` are run.
- Any non-zero value is considered true in C. Zero is considered false. So this statement is always true:

  ```
  if(1)
  ```

  And this statement is always false:

  ```
  if(0)
  ```
- You know whether a function returns a true or false value by reading the function's documentation, or you can set a true or false return value when writing your own functions.
- You cannot compare strings by using an `if` comparison. Instead, you use specific string comparison functions, which are covered in Chapter 13.
- When only one statement belongs to an `if` comparison, the braces are optional.

Exercise 8-3: Rewrite the code from Listing 8-1, removing the braces before and after Line 12. Build and run to ensure that it still works.

Comparing values in various ways

The C language employs a small platoon of mathematical comparison operators. I've gathered the bunch in Table 8-1 for your perusal.

Table 8-1 C Language Comparison Operators

Operator	*Example*	*True When*
!=	a != b	a is not equal to b
<	a < b	a is less than b
<=	a <= b	a is less than or equal to b
==	a == b	a is equal to b
>	a > b	a is greater than b
>=	a >= b	a is greater than or equal to b

Comparisons in C work from left to right, so you read *a >= b* as "a is greater than or equal to b." Also, the order is important: Both >= and <= must be written in that order, as must the != (not equal) operator. The == operator can be written either way. (See Listing 8-2.)

Listing 8-2: Values Are Compared

```
#include <stdio.h>

int main()
{
    int first,second;

    printf("Input the first value: ");
    scanf("%d",&first);
    printf("Input the second value: ");
    scanf("%d",&second);

    puts("Evaluating...");
    if(first<second)
    {
        printf("%d is less than %d\n",first,second);
    }
    if(first>second)
    {
        printf("%d is greater than %d\n",first,second);
    }
    return(0);
}
```

Exercise 8-4: Create a new project by using the source code shown in Listing 8-2. Build and run.

The most common comparison is probably the double equal sign. It may look odd to you. The == operator isn't the same as the = operator. The = operator is the *assignment operator,* which sets values. The == operator is the *comparison operator,* which checks to see whether two values are equal. (See Listing 8-3.)

I pronounce == as "is equal to."

Exercise 8-5: Add a new section to the source code from Listing 8-2 that makes a final evaluation on whether both variables are equal to each other.

Listing 8-3: Get "Is Equal To" into Your Head

```
#include <stdio.h>

#define SECRET 17

int main()
{
    int guess;

    printf("Can you guess the secret number: ");
    scanf("%d",&guess);
    if(guess==SECRET)
    {
        puts("You guessed it!");
        return(0);
    }
    if(guess!=SECRET)
    {
        puts("Wrong!");
        return(1);
    }
}
```

Exercise 8-6: Type the source code from Listing 8-3 into a new Code::Blocks project. Build and run.

Take note of the value returned by the program — either 0 for a correct answer or 1 for a wrong answer. You can see that return value in the Code::Blocks output window.

Knowing the difference between = and ==

One of the most common mistakes made by every C language programmer — beginner and pro — is using a single equal sign instead of a double in an `if` comparison. To wit, I offer Listing 8-4.

Listing 8-4: Always True

```
#include <stdio.h>

int main()
{
    int a;

    a = 5;

    if(a=-3)
    {
        printf("%d equals %d\n",a,-3);
    }
    return(0);
}
```

Exercise 8-7: Type the source code shown in Listing 8-4 into a new project. Run the program.

The output may puzzle you. What I see is this:

```
-3 equals -3
```

That's true, isn't it? But what happened?

Simple: In Line 9, variable *a* is assigned the value `-3`. Because that statement is inside the parentheses, it's evaluated first. The result of a variable assignment in C is always true for any non-zero value.

Exercise 8-8: Edit the source code from Listing 8-4 so that a double equal sign, or "is equal to," is used instead of the single equal sign in the `if` comparison.

Forgetting where to put the semicolon

Listing 8-5 is based upon Listing 8-4, taking advantage of the fact that C doesn't require a single statement belonging to an `if` comparison to be lodged between curly brackets.

Listing 8-5: Semicolon Boo-Boo

```
#include <stdio.h>

int main()
{
    int a,b;

    a = 5;
    b = -3;
```

```
    if(a==b);
        printf("%d equals %d\n",a,b);
    return(0);
}
```

Exercise 8-9: Carefully type the source code from Listing 8-5. Pay special attention to Line 10. Ensure that you type it in exactly, with the semicolon at the end of the line. Build and run the project.

Here's the output I see:

```
5 equals -3
```

The problem here is a common one, a mistake made by just about every C programmer from time to time: The trailing semicolon in Listing 8-5 (Line 10) tells the program that the `if` statement has nothing to do when the condition is true. That's because a single semicolon is a complete statement in C, albeit a null statement. To wit:

```
if(condition)
    ;
```

This construct is basically the same as Line 10 in Listing 8-5. Be careful not to make the same mistake — especially when you type code a lot and you're used to ending a line with a semicolon.

Multiple Decisions

Not every decision is a clean-cut, yes-or-no proposition. Exceptions happen all the time. C provides a few ways to deal with those exceptions, allowing you to craft code that executes based on multiple possibilities.

Making more-complex decisions

For the either-or type of comparisons, the `if` keyword has a companion — `else`. Together, they work like this:

```
if(condition)
{
    statement(s);
}
else
{
    statement(s);
}
```

When the *condition* is true in an `if-else` structure, the statements belonging to `if` are executed; otherwise, the statements belonging to `else` are executed. It's an either-or type of decision.

Listing 8-6 is an update of sorts to the code shown in Listing 8-1. The single `if` structure has been replaced by `if-else`. When the `if` comparison is false, the statement belonging to `else` is executed.

Listing 8-6: An if-else Comparison

```
#include <stdio.h>

int main()
{
    int a,b;

    a = 6;
    b = a - 2;

    if( a > b)
    {
        printf("%d is greater than %d\n",a,b);
    }
    else
    {
        printf("%d is not greater than %d\n",a,b);
    }
    return(0);
}
```

Exercise 8-10: Type the source code for Listing 8-6 into a new project. Compile and run.

Exercise 8-11: Modify the source code so that the user gets to input the value of variable *b*.

Exercise 8-12: Modify the source code from Listing 8-3 so that an `if-else` structure replaces that ugly `if-if` thing. (***Hint:*** The best solution changes only one line of code.)

Adding a third option

Not every decision made in a program is either-or. Sometimes, you find yourself in need of an either-or-or type of thing. In fact, no word is found in English to describe such a structure, but it exists in C. It looks like this:

```
if(condition)
{
    statement(s);
}
else if(condition)
{
    statement(s);
}
else
{
    statement(s);
}
```

When the first *condition* proves false, the `else if` statement makes another test. If that *condition* proves true, its statements are executed. When neither condition is true, the statements belonging to the final `else` are executed.

Exercise 8-13: Using the source code from Listing 8-2 as a base, create an `if-if else-else` structure that handles three conditions. The first two conditions are specified in Listing 8-2, and you need to add the final possibility using a structure similar to the one shown in this section.

C has no limit on how many `else if` statements you can add to an `if` decision process. Your code could show an `if`, followed by three `else-if` conditions, and a final `else`. This process works, though it's not the best approach. See the later section "Making a multiple-choice selection," for a better way.

Multiple Comparisons with Logic

Some comparisons are more complex than those presented by the simple operators illustrated earlier, in Table 8-1. For example, consider the following math-thingie:

```
-5 <= x <= 5
```

In English, this statement means that `x` represents a value between –5 and 5, inclusive. That's not a C language `if` comparison, but it can be when you employ logical operators.

Building a logical comparison

It's possible to load two or more comparisons into a single `if` statement. The results of the comparisons are then compared by using a logical operator. When the result of the entire thing is true, the `if` condition is considered true. (See Listing 8-7.)

Listing 8-7: Logic Is a Tweeting Bird

```
#include <stdio.h>

int main()
{
    int coordinate;

    printf("Input target coordinate: ");
    scanf("%d",&coordinate);
    if( coordinate >= -5 && coordinate <= 5 )
    {
        puts("Close enough!");
    }
    else
    {
        puts("Target is out of range!");
    }
    return(0);
}
```

Two comparisons are made by the `if` statement condition in Line 9. That statement reads like this: "If the value of variable *coordinate* is greater than or equal to –5 and less than or equal to 5."

Exercise 8-14: Create a new project using the source code from Listing 8-7. Build the program. Run the code a few times to test how well it works.

Adding some logical operators

The C language logical comparison operators are shown in Table 8-2. These operators can be used in an `if` comparison when two or more conditions must be met.

Table 8-2 Logical Comparison Operators

Operator	*Name*	*True When*
&&	and	Both comparisons are true
||	or	Either comparison is true
!	not	The item is false

Listing 8-7 uses the `&&` operator as a logical AND comparison. Both of the conditions specified must be true for the `if` statement to consider everything in the parentheses to be true.

Exercise 8-15: Modify the source code from Listing 8-7 so that a logical OR operation is used to make the condition true when the value of variable *`coordinate`* is less than –5 or greater than 5.

Exercise 8-16: Create a new project that asks for the answer to a yes-or-no question with a press of the Y or N key, either upper- or lowercase. Ensure that the program responds properly when neither a Y nor N is pressed.

- Logical operations are often referred to by using all caps: AND, OR. That separates them from the normal words *and* and *or*.
- The logical AND is represented by two ampersands: `&&`. Say "and."
- The logical OR is represented by two pipe, or vertical-bar, characters: `||`. Say "or."
- The logical NOT is represented by a single exclamation point: !. Say "not!"
- The logical NOT isn't used like AND or OR. It merely prefixes a value to reverse the results, transforming False into True and True into False.

The Old Switch Case Trick

Piling up a tower of `if` and `if-else` statements can be effective, but it's not the best way to walk through a multiple-choice decision. The solution offered in the C language is known as the switch-case structure.

Making a multiple-choice selection

The *switch-case structure* allows you to code decisions in a C program based on a single value. It's the multiple-choice selection statement, as shown in Listing 8-8.

Listing 8-8: Multiple Choice

```
#include <stdio.h>

int main()
{
    int code;

    printf("Enter the error code (1-3): ");
    scanf("%d",&code);

    switch(code)
    {
        case 1:
            puts("Drive Fault, not your fault.");
            break;
        case 2:
            puts("Illegal format, call a lawyer.");
            break;
        case 3:
            puts("Bad filename, spank it.");
            break;
        default:
            puts("That's not 1, 2, or 3");
    }
    return(0);
}
```

Exercise 8-17: Create a new project using the code from Listing 8-8. Just type it in; I describe it later. Build it. Run it a few times, trying various values to see how it responds.

Examine the source code in your editor, where you can reference the line numbers mentioned in the following paragraphs.

The `switch-case` structure starts at Line 10 with the `switch` statement. The item it evaluates is enclosed in parentheses. Unlike an `if` statement, `switch` eats only a single value. In Line 10, it's an integer that the user types (read in Line 8).

The `case` part of the structure is enclosed in curly brackets, between Lines 11 and 23. A `case` statement shows a single value, such as `1` in Line 12. The value is followed by a colon.

The value specified by each `case` statement is compared with the item specified in the `switch` statement. If the values are equal, the statements belonging to `case` are executed. If not, they're skipped and the next `case` value is compared.

The `break` keyword stops program flow through the `switch-case` structure. Program flow resumes after the `switch-case` structure's final curly bracket, which is Line 24 in Listing 8-8.

After the final comparison, the `switch-case` structure uses a *`default`* item, shown in Line 21. That item's statements are executed when none of the `case` comparisons matches. The *`default`* item is required in the switch-case structure.

Exercise 8-18: Construct a program using source code similar to Listing 8-8, but make the input the letters *A, B,* and *C.* You might want to review Chapter 7 to see how single characters are specified in the C language.

- The comparison being made in a `switch-case` structure is between the item specified in `switch`'s parentheses and the item that follows each `case` keyword. When the comparison is true, which means that both items are equal to each other, the statements belonging to `case` are executed.
- The `break` keyword is used to break the program flow. It can be used in an `if` structure as well, but mostly it's found in looping structures. See Chapter 9.
- Specify a `break` after a `case` comparison's statements so that the rest of the structure isn't executed. See the later section "Taking no breaks."

Understanding the switch-case structure

And now — presenting the most complex thing in C. Seriously, you'll find more rules and structure with `switch-case` than just about any other construct in C. Here's the skeleton:

```
switch(expression)
{
    case value1:
        statement(s);
        break;
    case value2:
        statement(s);
        break;
    case value3:
        statement(s);
        break;
    default:
        statement(s);
}
```

The `switch` item introduces the structure, which is enclosed by a pair of curly brackets. The structure must contain at least one `case` statement and the `default` statement.

The `switch` statement contains an *`expression`* in parentheses. That expression must evaluate to a single value. It can be a variable, a value returned from a function, or a mathematical operation.

A `case` statement is followed by an immediate value and then a colon. Following the colon are one or more statements. These statements are executed when the immediate value following `case` matches the `switch` statement's expression. Otherwise, the statements are skipped, and the next `case` statement is evaluated.

The `break` keyword is used to flee the `switch-case` structure. Otherwise, program execution falls through the structure.

The `default` item ends the `switch-case` structure. It contains statements that are executed when none of the `case` statements matches. Or, when nothing is left to do, the `default` item doesn't require any statements — but it must be specified.

The `case` portion of a `switch-case` structure doesn't make an evaluation. If you need multiple comparisons, use a multiple `if-else` type of structure.

Taking no breaks

It's possible to construct a `switch-case` structure with no `break` statements. Such a thing can even be useful under special circumstances, as shown in Listing 8-9.

Listing 8-9: Meal Plan Decisions

```
#include <stdio.h>

int main()
{
    char choice;

    puts("Meal Plans:");
    puts("A - Breakfast, Lunch, and Dinner");
    puts("B - Lunch and Dinner only");
    puts("C - Dinner only");
    printf("Your choice: ");
    scanf("%c",&choice);
```

```
    printf("You've opted for ");
    switch(choice)
    {
        case 'A':
            printf("Breakfast, ");
        case 'B':
            printf("Lunch and ");
        case 'C':
            printf("Dinner ");
        default:
            printf("as your meal plan.\n");
    }
    return(0);
}
```

Exercise 8-19: Create a new project using the source code from Listing 8-9. Build and run.

Exercise 8-20: If you understand how `case` statements can fall through, modify Exercise 8-18 so that both upper- and lowercase letters are evaluated in the `switch-case` structure.

The Weird ?: Decision Thing

I have one last oddball decision-making tool to throw at you in this long, decisive chapter. It's perhaps the most cryptic of the decision-making tools in C, a favorite of programmers who enjoy obfuscating their code. Witness Listing 8-10.

Listing 8-10: And Then It Gets Weird

```
#include <stdio.h>

int main()
{
    int a,b,larger;

    printf("Enter value A: ");
    scanf("%d",&a);
    printf("Enter different value B: ");
    scanf("%d",&b);

    larger = (a > b) ? a : b;
    printf("Value %d is larger.\n",larger);
    return(0);
}
```

Specifically, you want to look at Line 12, which I'm showing here as though it isn't ugly enough inside Listing 8-10:

```
larger = (a > b) ? a : b;
```

Exercise 8-21: Create a project using the source code from Listing 8-10. Build and run just to prove that the weirdo `?:` thing works.

Officially, `?:` is known as a *ternary* operator: It's composed of three parts. It's a comparison and then two parts: value-if-true and value-if-false. Written in plain, hacker English, the statement looks like this:

```
result = comparison ? if_true : if_false;
```

The statement begins with a comparison. Any comparison from an `if` statement works, as do all operators, mathematical and logical. I typically enclose the comparison in parentheses, though that's not a requirement.

When `comparison` is true, the `if_true` portion of the statement is evaluated and that value stored in the *`result`* variable. Otherwise, the `if_false` solution is stored.

Exercise 8-22: Rewrite the source code form Listing 8-10 using an `if-else` structure to carry out the decision and result from the `?:` ternary operator in Line 12.